# Efficient Incremental Breadth-Depth XML Event Mining

Rashed Salem, Jérôme Darmont and Omar Boussaïd
Université de Lyon (ERIC Lyon 2)
5 av. P. Mendès-France, 69676 Bron Cedex, France
{rashed.salem, jerome.darmont, omar.boussaid}@univ-lyon2.fr

## ABSTRACT

Many applications log a large amount of events continuously. Extracting interesting knowledge from logged events is an emerging active research area in data mining. In this context, we propose an approach for mining frequent events and association rules from logged events in XML format. This approach is composed of two-main phases: I) constructing a novel tree structure called Frequency XML-based Tree (FXT), which contains the frequency of events to be mined; II) querying the constructed FXT using XQuery to discover frequent itemsets and association rules. The FXT is constructed with a single-pass over logged data. We implement the proposed algorithm and study various performance issues. The performance study shows that the algorithm is efficient, for both constructing the FXT and discovering association rules.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

## General Terms

Algorithms, Performance

## Keywords

**keywords:** Mining logged events, XML mining, frequent itemsets, association rules.

## 1. INTRODUCTION

Recently, the eXtensible Markup Language (XML) has become widely used as the *de facto* standard for representing, exchanging, modeling, and maintaining semi-structured data. The widespread of XML-based applications and increasing amount of XML data pose several challenges for mining XML data. Modern XML-based applications log huge amounts of events at real-time, continuously. The logged event data describe the status of each application component and can be used to trace application activities. Applications that log events in XML format range from scientific to business and financial applications. Examples of such applications include XML-based data warehousing, web personalization and web-click logs, geographic information systems, and e-commerce. Mining and analyzing logged event from such applications help for achieving self-management systems. Therefore, mining XML-formatted logged events is becoming increasingly important. It should have high attention from the database, data warehousing, data mining, and machine learning research communities.

Mining logged events is the process of extracting knowledge from continuous, rapid logged events. One of the most important data mining techniques is *association rule mining*. Association rule mining discovers interesting association and/or correlation relationships among large sets of logged events, and predicts upcoming events based on occurrence of previous ones. Mining association rules from incremental XML-formatted logged events is different than mining traditional static data, due to several specific issues and challenges either related to data arrival [5, 7], or XML-formatting nature [4, 12].

When logging events, they arrive continuously at moderate or high speed, in unbounded amount, and changing data distributions. Unlike in traditional data mining, there is not enough time to rescan the whole database whenever an update occurs. Therefore, a single-pass over events is required. Logged events need to be processed incrementally as fast as possible. Processing speed should be faster than events arrival rate. Moreover, mined data should not need to be recalculated each time requested. Unbounded amount of logged events and limited system resources, such as disk storage, memory usage, and CPU power, lead to the need for event mining algorithms that adapt themselves to available resources, otherwise accuracy result decreases. Also, while traditional data mining techniques mine frequent itemsets and discard non-frequent itemsets, this property is not valid for logged events, where the frequency of itemsets is changing over time. On the other hand, extracting knowledge from XML data is more difficult than an operational data, because of the flexible, irregular, and semi-structured nature of XML data.

To the best of our knowledge, there is no algorithm proposed in the literature to discover interesting knowledge from incremental XML-formatted logged events. Therefore, we propose in this paper an incremental algorithm for this purpose. Our algorithm is composed of two main phases: firstly, we construct a new tree structure called Frequency

XML-based Tree (FXT) that stores frequencies of events to be mined. Secondly, we query frequent event-sets and association rules efficiently from the constructed FXT using XQuery. Our algorithm handles most processing logged event issues. It satisfies a single-pass over data transactions to construct the compact FXT structure. Although the FXT is processed using XML technologies and constructed in XML format, its construction time is fast enough. Association rules with different minimum supports are queried at any time without re-constructing the FXT from scratch.

The rest of this paper is organized as follows. Related work is discussed in section 2. In section 3, we present our motivation and a description of logged events. Section 4 introduces the general structure of the novel Frequency XML-based Tree (FXT) and our algorithm for constructing the FXT. Mining frequent itemsets and association rules from the FXT is presented in section 5. Performance study of our algorithm is discussed in section 6. Finally, we conclude and highlight future trends in section 7.

## 2. RELATED WORK

There are two main types of approaches for XML data mining in the literature. The first type of approaches applies relational data mining tools on XML data by mapping XML documents to relational data model and storing them in a relational database [11]. The second type of approaches applies data mining techniques directly onto native-XML data [2, 9, 10]. We are interested with the second type of approaches, specifically mining frequent itemsets and association rules from XML data.

### Mining association rules using XQuery.

Wan and Dobbie provide XQuery implementation of the well-known Apriori algorithm [1], to extract association rules from XML documents without any pre-processing or post-processing [9]. Their algorithm is adapted to simple and well-defined XML format. This algorithm is extended with pre-processing step in order to mine more complex and irregular XML documents [10]. Authors actually transform complex documents into a format that can be mined by Wan algorithm using XSLT. Braga et al. propose XMINE [2], a tool to extract XML association rules from XML documents. The XMINE operator is based on XPath and XQuery to express complex mining tasks on the content and the structure of XML data.

### Tree-based mining algorithms.

Han et al. propose FP-Growth for mining frequent itemsets without generating candidate itemsets [6]. FP-Growth requires two database scans for constructing its FP-Tree. Cheung and Zaiane extend FP-Tree by proposing a novel data structure called CATS Tree [3]. As FP-Tree, CATS tree allows frequent pattern mining without generation of candidate itemsets. It allows mining with a single pass over the database as well as efficient insertion or deletion of transactions at any time.

To the best of our knowledge, our algorithm is the first work proposed to mine frequent itemsets and association rules from incremental XML-formatted logged events, using XML technologies (e.g., XPath and XQuery). Table 1 shows differences between FXT versus tree-based techniques (i.e., FP-Growth and CATS) and XQuery-based implementation techniques (i.e., Apriori implementation). Although Apriori-implementation mines association rules from XML data using XQuery [9], it is designed to static transactions of XML data. Mining association rules with different minimum support by Apriori algorithm requires regenerating the largest itemsets from scratch. Compared to our algorithm, Apriori-implementation provides less performance particularly for large databases of transactions. Despite CATS [3] is not proposed for mining XML data, it is based on constructing an incremental frequency tree like our algorithm. Rather than CATS algorithm mines frequent patterns with a complicated algorithm named FELINE, it does not support mining association rules from CATS tree directly due to the absence of total size of transactions.

| | | FXT | FP-Growth | CATS | Apriori-imp. |
|---|---|---|---|---|---|
| Data context | *static* | | X | | X |
| | *incremental* | X | | X | |
| Data format | *raw data* | | X | X | |
| | *XML data* | X | | | X |
| Items coverage | *all items* | X | | X | |
| | *only frequent* | | X | | X |
| Tree structure | *tree-based* | X | X | X | |
| Passing database | *single-pass* | X | | X | |
| | *two or more pass* | | X | | X |
| Transactions size | *included in tree* | X | | | |
| Ordering tree | *pre-ordering* | X | | | |
| | *ordering path items w.r.t support* | X | X | X | |
| | *sorting children* | X | | X | |
| | *sweeping items* | | | X | |
| Implementation-based | *programming language* | | X | X | |
| | *XML technologies* | X | | | X |

**Table 1: FXT versus related works**

## 3. LOGGING EVENTS

There are several software platforms that log a large amount of events incrementally every day, into simple text or XML format. Logged events are essential to understand and trace the activities of such platforms. For instance, we are motivated to mine logged events from XML-based data integration platforms [8]. It worth to be noted that these platforms are developed, managed and maintained using XML technologies. Data integration is the process of extracting data from heterogeneous and distributed sources, transforming them into a unified format, and loading them into a repository (namely a warehouse), see Figure 1. Discovering interesting knowledge from logged events can be employed to self-maintain and configure the workflow behavior of these systems, how to achieve this issue is out scope of this paper.

Actually, logged events include much descriptive information about each activity (e.g., identification, occurring time, source, description, category, etc.). The more logging information, the more interesting knowledge can be discovered. In order to apply our algorithm for mining frequent events and association rules, we need to pre-process logged events and organize them into transactions as illustrated in the following sample.

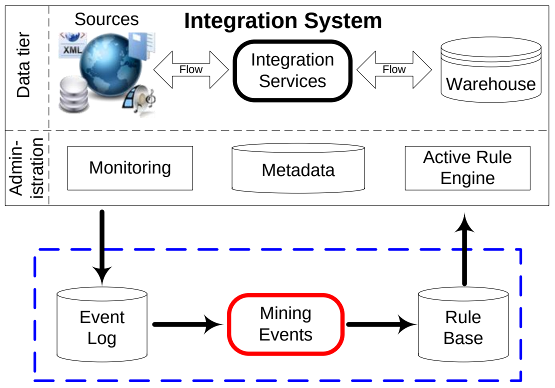


**Figure 1: Data integration system**

```
<transaction id="1" time="2011-04-10 09:16:00">
 <item>A</item> <item>B</item> <item>C</item> <item>D</item>
</transaction>
<transaction id="2" time="2011-04-10 09:16:20">
 <item>C</item> <item>E</item>
</transaction>
<transaction id="3" time="2011-04-10 09:16:40">
 <item>B</item> <item>C</item>
</transaction>
<transaction id="4" time="2011-04-10 09:17:00">
 <item>C</item> <item>D</item> <item>E</item>
</transaction>
<transaction id="5" time="2011-04-10 09:17:20">
 <item>B</item> <item>C</item> <item>D</item>
</transaction>
<transaction id="6" time="2011-04-10 09:17:40">
 <item>A</item> <item>C</item> <item>E</item>
</transaction>
```

Each transaction has identification, its occurring time, and a set of items which represent platform events. The set of events is assumed to be logged in a window-size of time (time + window). The corresponding format of logged transactions can be obtained directly from their origin platforms, or can be transformed using the XSLT language to a format that can be processed by our algorithm in a preprocessing step. The most important thing to our algorithm is to define the listing of items of each transaction, which should be sorted alphabetically for performance purposes.

## 4. FREQUENCY XML-BASED TREE (FXT)

### 4.1 FXT Structure

In order to mine frequent itemsets or association rules, the frequency of events (or items) needs to be calculated. Hence, we propose a novel tree structure that contains frequency of all logged items, named Frequency XML-based Tree (FXT). The FXT nodes, except root node, consist of two entries: *item name* and *counter*, where *item name* registers which item this node represents (e.g., $I_i$), and *counter* registers the number of transactions represented by the portion of the path reaching this node (e.g., $N_i$ or $N_{m|\ldots|i}$). As illustrated in figure 2, the FXT is composed of three main levels of nodes. Firstly, the *Root node* refers to the FXT root node. It represents the total number of logged transactions ($N_{trans}$). Secondly, the *Breadth nodes* refers to all root's children nodes. It represents the count of each item appeared in any logged transaction. Thirdly, the *Depth nodes* refers to all root's grandchildren nodes. It represents a relative or conditional count of a specific item given other related items. The depth nodes are represented as set of paths, each path corresponds specific transactions itemsets. In figure 2, the dashed line annotated by double slashes "//" means that there is zero or more in-between nodes in a specific depth path.

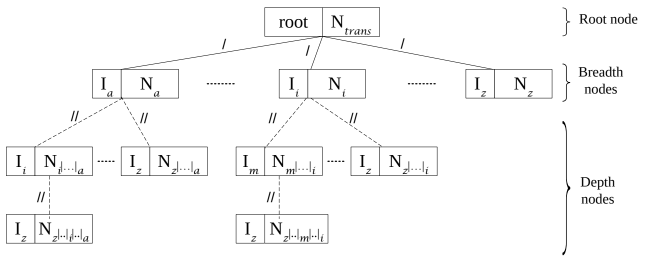


**Figure 2: FXT structure**

It worth to be noted that the FXT can handle both sorted and unsorted items of upcoming transactions, but we observed that handling sorted items results in more compact FXT structure and eases mining frequent itemsets and association rules from the FXT. Thus, letters ($a$, $i$, $m$, and $z$) of items refer to their ordering. In addition, although FXT is designed to manage XML-formatted data, the same concept can be applied to raw data. Finally, there are some facts can be deduced from the FXT structure:

- $N_{trans} = Total(trans)$ refers to the total number of transactions;
- $N_{trans} \geq N_k$, where $N_k$ can be count of any item $k$;
- $N_k \geq N_{v|\ldots|k}$, where $N_{v|\ldots|k}$ is a conditional count of $I_v$ given $I_k$ and in-between items.

### 4.2 FXT Management

The first phase of our algorithm is to construct the FXT, by handling each logged transaction individually.

#### *4.2.1 Insertion of transactions*

Logged transactions are inserted into the FXT upon arrival. Our algorithm follows four steps for each logged transaction on constructing the FXT as presented by algorithm 1.

**Step 1** *(incrementing root counter, $N_{trans}$).*
This root counter represents the total number of logged transactions, which can be used to calculate item support.

**Step 2** *(incrementing breadth).*
For each item of the transaction, our algorithm increments the item counter if it exists as one of root children (breadth nodes), otherwise the algorithm creates the item as new root child and initializes its counter at 1. Any *item* support can be easily calculated later via dividing item counter by $N_{trans}$, see algorithm 2.

**Step 3** *(incrementing depth).*
The algorithm increments the transaction path if it exists, otherwise creates it. While creating the path, item by

```
Algorithm 1: FXT construction
  Input: Set of transactions (S)
  Output: FXT document
  begin
      foreach $T \in S$ do
          idx = 0
          (: step1: increment root counter :)
          root/@counter++
          increment-or-create-breadth(T)
          increment-or-create-depth(T)
          update-other-paths(T, idx )
      end
  end
```

```
Algorithm 2: Incrementing breadth
  Procedure: increment-or-create-breadth(T)
  Input: Transaction(T)
  begin
      foreach $item \in T$ do
          if $item \in root/*$ then
              item/@counter++
          else
              (: create new item as root child, initialize its
              counter at 1 :)
              insert root/item          (: item as root child :)
              item/@counter=1
          end
      end
  end
```

item, respecting the transaction items ordering, the algorithm takes into account the previous occurrence of relative transaction items. This is reflected when initiating counter of the path items. The FXT path may not correspond only the same transaction that occurred once or several times, but also correspond many transactions that satisfy the same beginning portion of the path. This step is presented by algorithm 3.

**Step 4** *(updating other paths)*.

This step is required to ensure the correctness of counting of one given itemset across different FXT paths. For each transaction, some paths can be generated from transaction items that differ from the path built in step 3, called *other paths*. The algorithm checks only other paths existing in the FXT to be updated. In case if they do not already exist, the algorithm does not create them for compactness purpose, see algorithm 4.

### 4.2.2 Example

Figure 3(a-f) shows the four steps to construct the FXT by inserting transactions given in section 3. Because steps 1 and 2 are always applied directly for all transactions, we focus on how steps 3 and 4 are applied.

In figure 3(a) and figure 3(b), step 3 creates the paths "root/A/B/C/D" and "root/C/E", respectively. Step 4 is not evaluated, because there are no other paths available. In figure 3(c), in order to initialize counter of item "C" according to step 3, the algorithm detects item "C" as child of item "B" in the path "root/A/B/C/D". Thus, counter of item "C" in the existing path is incremented to become an initialization value of item "C" in the new path "root/B/C". In figure 3(d), step 3 initializes counter of item "D" at 2, because item "D" already exists as child of item "C" in the path "root/A/B/C/D". But, step 4 detects other path "root/C/E" in the FXT, thus the counter of item "E" is incremented. In figure 3(e), step 3 initializes the counter of item "D" at 2, because item "D" already exists as grandchild or child of items "B" and "C", respectively in the path "root/A/B/C/D". Moreover, step 4 detects portion of other path "root/C/D" in the FXT, thus counter of item "D" is incremented. In fig-

```
Algorithm 3: Incrementing depth
  Procedure: increment-or-create-depth(T)
  Input: Transaction(T)
  begin
      path = root
      preItemPaths = 0
      for idx = 0 to length(T)-1 do
          path ← path / item[idx]          (:where item is T
          member:)
          nexIdx ← idx+1
          nextItem ← item[nexIdx]          (:next item in T:)
          (: paths that have the item anywhere: )
          preItemPaths ← preItemPaths // item[idx]
          (:if inserted item is child of its previous item path:)
          if $nextItem \in path/*$ then
              nextItem/@counter++
              (: if next item is descendant of its previous
              items :)
          else if $nextItem \in root//preItemPaths//*$ then
              insert path/nextItem
              (: max preItem counter is incremented as
              initialization counter :)
              nextItem/@counter ←
              Max(root//preItemPaths//nextItem/@counter)+1
          else
              insert path/nextItem
              nextItem/@counter=1
          end
      end
  end
```

```
Algorithm 4: Updating other paths
  Procedure: update-other-paths(T, idx )
  Input: Transaction(T), index of breadth item (idx)
  begin
      path = root
      for idx to length(T)-1 do
          path ← path / item[idx]       (: where item is T
          member :)
          (: skip 1st-then-2nd item sequence to generate other
          possible paths :)
          if idx = 0 then
              nexIdx = 2
          else
              nexIdx = idx+1
          end
          for nexIdx to length(T) do
              leafItem ← item[nexIdx]
              (:if leaf item is descendant of its previous items:)
              if $leafItem \in path/*$ then
                  leafItem/@counter++
              end
          end
          (: repeat starting with next item of T as parent
          of the path :)
          update-other-paths(T, nexIdx)
      end
  end
```

ure 3(f), step 3 initializes the counter of item "C" at 2, because item "C" already exists as grandchild of item "A" in the path "root/A/B/C/D". Also, step 4 detects other path "root/C/E" in the FXT, thus counter of item "E" is incremented. Finally, the constructed FXT is as follows.

```
 <? xml version="1.0" >
 <root counter = "6">
   <A counter = "2">
     <B counter = "1">
       <C counter = "1">
         <D counter = "1"/>
       </C>
     </B>
     <C counter = "2">
      <E counter = "1"/>
     </C>
  </A>
  <B counter = "3">
    <C counter = "3">
      <D counter = "2"/>
    </C>
  </B>
  <C counter = "6">
    <E counter = "3"/>
    <D counter = "3">
      <E counter = "1"/>
    </D>
  </C>
  <D counter = "3"/>
  <E counter = "3"/>
</root>
```

## 5. MINING FREQUENT ITEMSETS AND ASSOCIATION RULES USING XQUERY

The main objective of constructing the FXT is to mine frequent itemsets and association rules easily using the XQuery language. Frequent itemsets are queried by traversing the FXT from breadth nodes to specific nodes (portion of paths), or to leaf nodes (complete paths). Frequent itemsets are filtered using a statistical measure called *support*. Support measures the proportion of transactions that contains a specific item (or itemset). A frequent itemset is an itemset whose support is greater than some user-specified minimum support. Frequent itemsets satisfy the Apriori property, which states that if a given portion of path does not satisfy minimum support, then neither will any of its descendants [1]. Examples for retrieving the support of items and itemset from the FXT follow.

$$Support(A) = \frac{root/A/@counter}{root/counter}$$

$$Support(B,C,D) = \frac{root/B/C/D/@counter}{root/counter}$$

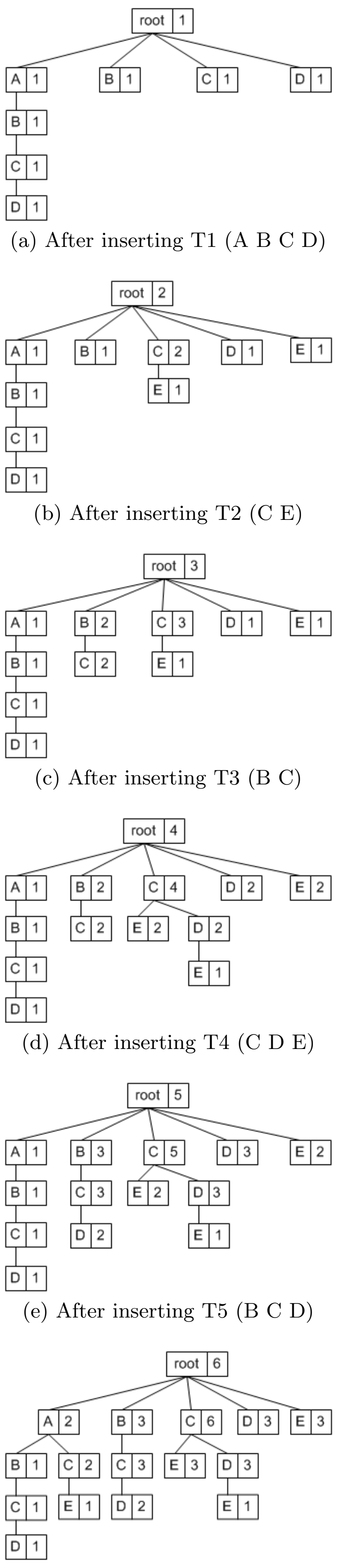
***Example:*** The following example introduces the function for generating frequent itemsets from the FXT that constructed in main example (section 4.2.2).

```
declare variable $input := doc("tree.xml")/root;
declare variable $rootCounter := $input/@counter;

declare function local:getFrequentItemsets($parent as xs:string,
$element as element(*, xs:untyped), $minSupport as xs:decimal) {

let $path := concat($parent,'/',name($element))
 where $element/@counter div $rootCounter>=$minSupport
return
 (<frequent path="{$path}" count="{$element/@counter}"
  support="{$element/@counter div $rootCounter}"/>,

 for $child in $element/*
```



(a) After inserting T1 (A B C D)

(b) After inserting T2 (C E)

(c) After inserting T3 (B C)

(d) After inserting T4 (C D E)

(e) After inserting T5 (B C D)

(f) After inserting T6 (A C E)

**Figure 3: Constructing the FXT document.**

```
 return
  local:getFrequentItemsets ($path, $child, $minSupport)) };

(: call the function :)
for $child in $input/*
return
 local:getFrequentItemsets("", $child, 0.25)
```

The result of calling previous function to get frequent itemsets with min support=0.25 is as follows.

```
<frequent path="/A" count="3" support="0.375"/>
<frequent path="/A/B" count="2" support="0.25"/>
<frequent path="/A/B/D" count="2" support="0.25"/>
<frequent path="/A/C" count="2" support="0.25"/>
<frequent path="/B" count="4" support="0.5"/>
<frequent path="/B/C" count="3" support="0.375"/>
<frequent path="/B/C/D" count="2" support="0.25"/>
<frequent path="/C" count="6" support="0.75"/>
<frequent path="/C/E" count="3" support="0.375"/>
<frequent path="/C/D" count="3" support="0.375"/>
<frequent path="/D" count="4" support="0.5"/>
<frequent path="/E" count="4" support="0.5"/>
```

Association rules have been first introduced in the context of retail transaction databases [1]. An association rule is an implication of the form $X \Rightarrow Y$, where the rule *body* $X$ and *head* $Y$ are subsets of the set $I$ of *items* $(I = i_1, i_2, ..., i_n)$ within a set of *transactions* $D$ and $X \cap Y = \Phi$. A rule $X \Rightarrow Y$ states that the transactions $T$ that contain the items in $X$ are likely to also contain the items in $Y$. Association rules are characterized by two measures: the *support*, which measures the proportion of transactions in $D$ that contain both items $X$ and $Y$; and the *confidence*, which measures the proportion of transactions in $D$ containing items $X$ that also contain items $Y$. $Confidence(X \Rightarrow Y)$ can be expressed as the conditional probability $p(Y|X)$. Thus, we define:

$$Support(X \Rightarrow Y) = \frac{count(X \cup Y)}{N_{trans}} = \frac{root/X/Y/@counter}{root/counter} \quad (1)$$

$$Confidence(X \Rightarrow Y) = \frac{support(X \Rightarrow Y)}{support(X)} = \frac{count(X \cup Y)}{count(X)} = \frac{root/X/Y/@counter}{root/X/@counter} \quad (2)$$

***Example:*** The following example introduces the XQuery function for generating a set of association rules from the FXT constructed in main example (section 4.2.2).

```
declare function local:generateRules($parent as xs:string,
$x as element (*, xs:untyped), $min_sup as xs:decimal,
$min_conf as xs:decimal) {

let $path_x := concat($parent,'/',name($x))
return
 (for $y in $x/*
  let $y_given_x := name($y)
  let $support_xy := $y/@counter div $rootCounter
  let $support_x := $x/@counter div $rootCounter
  let $confidence := $support_xy div $support_x
  where $support_xy >= $min_sup and $confidence >= $min_conf
 return
  (<rule body="{$path_x}" head="{$y_given_x}"
        support="{$support_xy}" confidence="{$confidence}"/>,
  local:generateRules($path_x, $y,$min_sup, $min_conf)))  };

(: call the function :)
for $child in $input/*
return
 local:generateRules("", $child, 0.25, 0.5)
```

The result of calling previous function to get association rules with min support=0.25 and min confidence=0.5 is as follows.

```
<rule body="/A" head="B" support="0.25" confidence="0.666"/>
<rule body="/A/B" head="D" support="0.25" confidence="1"/>
<rule body="/A" head="C" support="0.25" confidence="0.666"/>
<rule body="/B" head="C" support="0.375" confidence="0.75"/>
<rule body="/B/C" head="D" support="0.25" confidence="0.666"/>
<rule body="/C" head="E" support="0.375" confidence="0.5"/>
<rule body="/C" head="D" support="0.375" confidence="0.5"/>
```

Note that, it is possible to apply other XQuery functions to discover some statistics or mine more association rules. For instance, to query the reverse $rule(Y \Rightarrow X)$, a function is firstly required to sort the rule body and the rule head alphabetically in order to calculate the $support(Y \Rightarrow X)$, whereas
$count(Y \cup X) = count(X \cup Y) = root/x/y/@counter$. But, when calculating the $confidence(Y \Rightarrow X)$, the rule body does not change in the denominator, i.e., $support(Y)$, see equation (2).

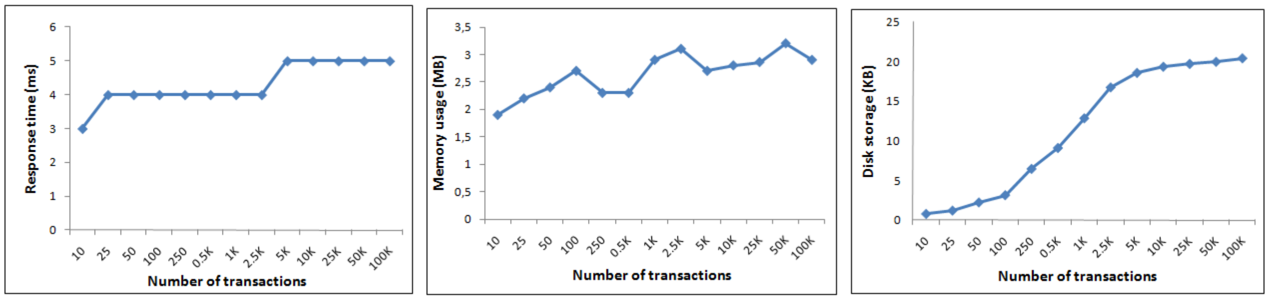
## 6. PERFORMANCE STUDY

We have implemented the FXT construction algorithm using some Java libraries for manipulating XML data structure (i.e., JDom, SAXPath, and Jaxen). Mining frequent itemsets and association rules are performed using the XQuery language. We experimented with different synthetic datasets, starting from 10 transactions to 100K of transactions. The average lengths of transactions are 15 items per transaction. All experiments are performed on a 2.80 GHz PC with 3 GB RAM, running on Windows 7, with minimum Java heap size 128 MB and maximum Java heap size 512 MB.

We study the impact of constructing the FXT on the machine resources. Figure 4(a) plots CPU time for new transaction insertion given different FXT sizes. It can be easily observed that the CPU runs fast for inserting new transaction even though FXT has large size (e.g., it takes 5ms to insert new transaction into a 100K FXT size). Likewise, figure 4(b) plots memory usage, it can be observed that our algorithm consumes a small size of memory for new transaction insertion with different FXT sizes. Figure 4(c) plots disk storage of the FXT document against different sizes of transactions. As shown in the figure, although the increasing relationship, the required storage remains small. Due to the FXT compact structure, the repeated or similar insertions of transaction need to only update item counters without consuming further storage space.

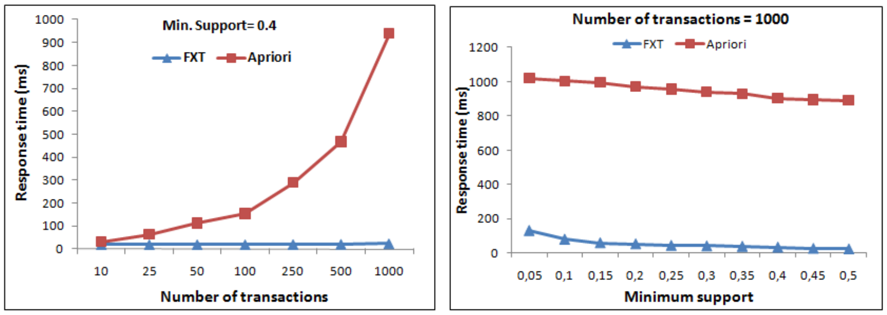
Since we are interested in mining XML data using XML technologies, to the best of our knowledge there is only one most related work (i.e., implementation of Apriori algorithm using XQuery [9]). The Apriori algorithm always deals with static database of transactions. Figure 5 shows the performance comparison between our algorithm and the XQuery-based implementation of Apriori, for mining association rules from XML using XQuery. It shows that our algorithm is always providing better performance than Apriori, specifically for larger amount of transactions (see figure 5(a)), and also for different values of minimum support (see figure 5(b)). Apriori generates frequent itemsets and association rules each time from scratch, while our algorithm construct the FXT incrementally. Then frequent itemsets and association rules can be queried directly at any time from the FXT.Moreover, FXT is very compressed if compared with transactions document of Apriori algorithm.

Finally, we conclude that our algorithm is very efficient to



(a) (b) (c)

**Figure 4: Machine resources for inserting new transaction to FXT**



(a) (b)

**Figure 5: Performance comparison between FXT and Apriori algorithm**

consume resources. It can also mine frequent itemsets and association rules against different support and confidence values, without reconstructing its FXT from scratch that results in a better performance. Additionally, FXT performance is better than XQuery-based Apriori implementation.

## 7. CONCLUSIONS

In this paper, we propose an incremental approach for mining association rules from XML logged events. Our approach applies an incrementing breath-then-depth algorithm, for constructing a novel frequency XML-based tree structure. The algorithm composes of four steps for inserting transaction into the tree. The constructed tree can be directly queried using XQuery language for retrieving frequent itemsets and association rules, without applying complex data mining techniques. Our algorithm handles incremental logged events. Thus, it is featured with a single-pass of dataset, incremental processing of transaction, compressed structure of the tree, fast for inserting new transactions, fast for querying frequent itemsets or association rules, and efficient to limited resources. These features are validated by implementing the algorithm and experimenting its performance.

In future, we aim at mining association rules from logged events taking into account their real-time of logging, and discovering the relationships among events against their logged real-time. Moreover, we intend to apply our algorithm for mining XML events that logged from our data integration platform [8]. This algorithm can be used to discover interesting knowledge, in order to maintain, automate, and re-activate the workflow behavior of the ETL tasks.

## 8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Very Large DataBase, VLDB*, pages 487–499, 1994.

[2] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P. L. Lanzi. A tool for extracting `XML` association rules. *Proceedings of 14th IEEE International Conference on Tools with Artificial Intelligence, ICTAI*, pages 57–64, 2002.

[3] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support constraint. *7th International of Database Engineering and Applications Symposium, IDEAS*, pages 111–116, 2003.

[4] L. Feng and T. Dillon. Mining interesting XML-enabled association rules with templates. Springer, 2004.

[5] M. M. Gaber, A. B. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, 2005.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 29(2):1–12, 2000.

[7] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *ACM SIGMOD Record*, 35(1):14–19, 2006.

[8] R. Salem, O. Boussaid, and J. Darmont. Conceptual workflow for complex data integration using `AXML`. In *International Conference on Machine and Web Intelligence (ICMWI 10)*, October 2010.

[9] J. W. W. Wan and G. Dobbie. Mining association rules from `XML` data using `XQuery`. *In Proc. Australasian Workshop on Data Mining and Web Intelligence, DMWI*, pages 169–174, 2004.

[10] X. Wang and C. Cao. Mining association rules from complex and irregular `XML` documents using `XSLT` and `XQuery`. *International Conference on Advanced Language Processing and Web Information Technology, ALPIT*, pages 314–319, 2008.

[11] J. Zhang, H. Liu, T. W. Ling, R. M. Bruckner, and A. M. Tjoa. A framework for efficent association rule mining in xml data. *Journal of Database Management (JDM)*, 17(3):19–40, 2006.

[12] Q. Zhao, L. C. 0002, S. S. Bhowmick, and S. K. Madria. XML structural delta mining: Issues and challenges. *Data Knowl. Eng*, 59(3):627–651, 2006.